Coordination Tools for MAS Development and Deployment

Enrico Denti
DEIS, Università di Bologna
Viale Risorgimento 2, 40136 Bologna, Italy
http://lia.deis.unibo.it/~ed
mailto:edenti@deis.unibo.it

Andrea Omicini
DEIS, Università di Bologna
Via Rasi e Spinelli 176, 47023 Cesena, FC, Italy
http://lia.deis.unibo.it/~ao
mailto:aomicini@deis.unibo.it

Alessandro Ricci
DEIS, Università di Bologna
Via Rasi e Spinelli 176, 47023 Cesena, FC, Italy
http://lia.deis.unibo.it/~ar
mailto:aricci@deis.unibo.it

Abstract

One of the most critical issues in the engineering of multiagent systems (MAS) is the inadequacy of the available tools for MAS development and deployment. As we assume interaction as a first-class issue in MAS, tools are particularly required to monitor and debug *inter-agents* aspects, such as interaction protocols, coordination policies, social norms and environment constraints. Since we claim that the definition of such tools is a basic research issue, in this paper we aim at identifying the main requirements for development and deployment tools within an effective agent infrastructure. Focusing on agent interaction aspects, we take tuple-based coordination infrastructures, – in particular the TuCSoN technology and tools – as our reference, and discuss the role of tools in a simple case study: the development and deployment of a well-known agent interaction protocol, the Contract Net.

1 Coordination Issues in Agent-based Development

A *multiagent system* (MAS) is more than just a sum of individual agents: how agents interact with other agents and the environment, and how they are organised in an ensemble, cannot be reasonably handled as individual issues. In this respect, *interaction* should be considered a fundamental design dimension – the dimension of the social aspects that determine collective behaviour. Therefore, interaction should not be treated as a mere afterthought following the design of the individual system components: instead, it is indeed a critical design issue, to be accounted for since the earliest design phases.

But what does "taking interaction into account" actually mean? Just defining and implementing a set of interaction protocols – whatever mechanism is used – is not enough, since this would typically not ensure that the agent ensemble behaves as desired, respecting global

constraints and ensuring the achievement of the desired global system. Instead, exploiting interaction as a crucial design dimension requires the ability to *govern* it in a precise, well-specified, and controlled way. Yet, mastering interaction is much a less established area (Wegner 1997).

Constraining and governing interaction is precisely the purpose of *coordination models* and *languages* (Gelernter & Carriero 1992): in the same way as agent languages and architectures provide abstractions and tools to design and build the individual agents, coordination models, languages and architectures provide abstractions and tools to manage and rule the agent interaction space. Effectively governing interaction basically means the ability to specify and possibly enforce goals, constraints, and desired properties that are not peculiar to a given agent, but to an ensemble of agents, and also the ability to adapt / modify this "social glue" dynamically, at run-time – all, without affecting agent autonomy. While computation languages express the inner working algorithm of an agent, coordination languages express the agent's observable behaviour – what is needed to design its interaction protocol.

Dependencies in MAS can be classified in two classes (Schumacher 2001), *objective* and *subjective*, basically according to the viewpoint over the space of agent interaction that is adopted. Objective dependencies refer to *inter-agent* dependencies, and typically concern the configuration of the system in terms of the basic interaction media, agent generation / destruction, and environment organisation; subjective dependencies, instead, refer to the *intra-agent* dependencies toward other agents. The management of subjective dependencies is called *subjective coordination* and is essentially concerned with intra-agent aspects. The management of objective dependencies, instead, is called *objective coordination*, and is essentially concerned with inter-agent aspects, since the dependencies are external to agents. So, in the subjective case, coordination comes from the attitudes of each individual toward the organisations/societies it belongs to, while objective coordination prescinds from the subjective view of the coordinated agents, and promotes the separation between the individual perception of coordination and the global coordination issues. As a result, objective coordination enables the modelling and shaping of the interaction space somehow independently of the interacting entities.

However, just providing a suitably-expressive (objective or subjective) coordination model is not enough to effectively support the engineering of Internet-based systems: suitable coordination technologies and run-time support should be made available as well. An efficient design, development and deployment process requires suitable *infrastructures*, providing designers and developers with a reliable support for commonly-required application services: for the above reasons, *coordination services* should be regarded as fundamental ones. To work with the widest range of hardware devices and software platforms, including PDAs and embedded systems, and to support the widest set of applications as well, a *coordination infrastructure* for agent-based applications should be easily deployable, light-weight, and both statically and dynamically configurable. Also, it should be Internet-compliant, and come along with effective *methodologies* and *tools* supporting the *design*, *development*, and *debugging / monitoring* of interaction.

In order to be effective, such methodologies and tools should fully support and promote the coordination model endorsed by the infrastructure: in particular, the tools for system development and deployment should be inspired by, and modelled after, the metaphors and abstractions of the coordination model – otherwise, the gap between theory and practice would likely make the design, development and deploying task basically unmanageable for any non-trivial system. We claim that the definition of such methodologies and tools is a fundamental research issue: it is the researchers' responsibility to analyse and indicate how the coordination model's concepts and run-time model should transpose onto suitable methodologies and tools, so as to ensure fundamental properties for system engineering – incremental development, system verification, debugging and maintenance, dynamic configurability, etc.

2 Deployment Tools for Interaction

Here we focus on run-time environments and tools for the support of deployment (from debugging to testing, from monitoring to analysis and run-time verification) of the interaction-related aspects of MAS. While literature on tools and toolkits for the design and development of agent systems and agent infrastructures abounds, very little seems to be available concerning deployment tools (Ndumu, Nwana, Lee & Collis 1999). Perhaps, quoting Van Liederkerke & Avouris (1995), this is due to the fact that *"designing and building MAS is a complex process: the integrated system debugging phase (...) presents particular difficulties (...) from the well-documented limitations that a human faces when observing and monitoring a collection of distributed concurrent processes"* (Garcia-Molina, Germano & Kohler 1989). Yet, the need for tools supporting the deployment of complex systems is clearly recognised: Rover, Waheed, Mutka & Bakic (1998) propose a classification of the most typical tools, categorised in debugging, testing, verification tools, etc.

2.1 Debugging Tools in Concurrent / Parallel Programming

Debugging a MAS is known to be a complex activity, aimed at identifying *structural* and *functional* errors (Ndumu et al. 1999): structural errors concern the organisation, structure and relationships between agents, while functional errors involves mistakes in the logic of the tasks to be performed. There is, however, one third class of errors, the *coordination errors*, that occur when the individual agent behaviour is structurally and functionally correct, but the emergent behaviour of the overall system is incorrect: this is precisely where a suitable coordination infrastructure, providing *ad hoc* tools, can be of help.

Typically, traditional debugging tools in concurrent and distributed programming provide features such as *breakpoints* on the single process, *traps* to catch process exceptions, single-step *tracing*, and explicit read/write of selected memory location or CPU registers; others go further, enabling traps on inter-process communication, the ability to add/remove inter-process messages, and the chance of controlling timeouts (McDowell & Helmbold 1989, Garcia-Molina et al. 1989). The most recent and advanced debugging/monitoring tools are event-based (Bates 1995): by catching event sequences, they make it possible to post-analyse and post-process event histories, so as to reconstruct the distributed system behaviour.

However, it is known (McDowell & Helmbold 1989) that observing the execution of concurrent processes is not a neutral operation, since it can interfere with the system behaviour (*probe effect*). This problem can be easily understood, since the addition of any spy, print or trace operation alters the timing of a process. So, for instance, critical races, which could be the cause of the system malfunctioning, and the very reason for debugging, may disappear in the new "debug-oriented" system configuration (Ndumu et al. 1999).

These problems are due to the lack of separation between computation and interaction: the *state of communication* is not explicitly represented, and so can not be observed without affecting the actors of communication – which is precisely what changes irremediably their original behaviour. As a general principle, observing a given level with no strong interferences requires an explicit representation of that level and, consequently, a *meta level* observer. This does not mean that all interferences can be prevented at all: however, their impact is reduced, since they can occur only at the meta level.

2.2 Mediated Interaction

In mediated interaction models, interaction never occurs directly between interacting entities: communication occurs via some kind of *communication abstraction* explicitly introduced for the purpose of enabling communication. A typical form of mediated interaction is *blackboard*-based interaction (Englemore & Morgan 1988): there, interacting entities write/read messages to/from

the blackboard only. This approach makes it possible to achieve some fundamental properties, such as *name uncoupling*, *space uncoupling* and *time uncoupling* – that is, interacting entities do not necessarily have to know each other explicitly, nor to be in the same place, not even to co-exist at the same time, in order to communicate. Moreover, the set of messages in the blackboard represents at any time the *state of communication*, which can therefore be observed quite simply.

In tuple-based interaction (Rossi, Cabri & Denti 2001), originated by the Linda model (Gelernter & Carriero 1992), the communication medium is the *tuple space*, that is, a multi-set of *tuples*, which are ordered collections of data chunks. Agents still communicate by adding (`out`), removing (`in`) and reading (`rd`) tuples to/from the tuple space, but access is associative, based on some matching mechanism like pattern matching or unification.

The existence of explicit communication media raises interaction to a first-class issue: from the model to the implementation, interaction is promoted to deserve explicit abstractions and places (blackboards or tuple spaces) devoted to it – and to its own handling. This key difference with respect to the previous approaches is what makes the communication state not only explicitly available, but also observable *without affecting* the communicating entities: in fact, since all communication acts occur in given place(s), putting such places under observation requires no changes on the original system configuration, avoiding the probe problem at that level a-priori.

2.3 Coordination and Tuple Centres

While communication means enabling interaction, coordination means constraining and ruling interaction (Wegner 1996). In Linda, coordination laws are built-in the tuple spaces, since the primitive operations and their semantics are pre-defined and fixed. If such laws are inadequate to the application, changing the tuple space behaviour is impossible: the desired overall behaviour must be built upon the fixed tuple space behaviour, charging agents to "fill the gap" – thus violating the separation between computation and coordination.

To overcome this limitation, one must either add new primitives to the basic tuple space model, or change the semantics of the existing operations as needed (Rossi et al. 2001). *Tuple centres* (Omicini & Denti 2001*b*) are tuple spaces whose behaviour can be defined by means of *reactions* to communication events, expressed in the Turing-equivalent ReSpecT language (Omicini & Denti 2001*a*): a tuple centre can then be programmed so as to feature potentially any desired behaviour.

Tuple centre behaviour is expressed in terms of ReSpecT *specification tuples*: correspondingly, a tuple centre can be thought as made of two parts – the *tuple space*, holding ordinary data tuples, and the *specification space*, holding specification tuples. So, both the state of communication and the *state of coordination*, expressed uniformly as logic tuples, are fully observable and modifiable. *Ad hoc* primitives are also provided to add, consult and remove specification tuples from the specification space.

Formally, given the set S of specification tuples, a tuple centre state is described at any time by the set T of ordinary tuples, the set W of the currently-pending queries, and the set Z of triggered reactions, waiting to be executed.[1] In particular, if S is empty, the tuple centre behaviour defaults to the standard tuple space behaviour. These sets provide different relevant views over interaction. The current communication state is captured by T and W: more precisely, T, being concerned with the data exchanged among agents, provides a *data-oriented* view over communication, whereas W, being concerned with the act of communication, provides a *control-oriented* view over communication. The current coordination state is instead represented by S and Z: S, contaning the static specification of the coordination rules, provides a *static view* over coordination, whereas Z, along with the execution step of the tuple centre

[1] For details on tuple centre and ReSpecT semantics, we forward the interested reader to (Omicini & Denti 2001*b*) and (Omicini & Denti 2001*a*), respectively.

virtual machine, represents the dynamic enactment of coordination laws, and provides a *dynamic* view over coordination.

2.4 Deployment with Mediation

When interaction is mediated by some kind of explicit communication media, observing interaction no longer requires changes to the interacting entities, since the communication state is represented and observable in the communication medium. This chance does not mean, however, that any interference with the system is automatically prevented: in standard tuple spaces, for instance, the insertion of a given tuple can only be observed by explicitly checking this kind of event by having an agent performing a suspensive matching `rd`. So, observation still takes place at the same level where communication occurs: this holds even if the whole tuple set is retrieved via a sort of `rdAll` or `copyAll` bulk primitive, which provides a snapshot, at that given time, of the tuple space state. So, the interaction histories of this communication medium change with respect to the case where the observer agent is not present. Also, some communication events may not be observable by definition: in standard tuple spaces `rd` operations cannot be observed, since they cause no change to the communication state (i.e., to the set of tuples). Even the `out` operation, dependently on the implementation, may not be observable, if an agent suspended on a matching `in` operation removes the tuple before it is actually added to the tuple space.[2].

Though mediated interaction does not guarantee absence of interferences, yet it is the key to reduce such interferences, bounding them inside an upper (meta) level. In tuple centres, in particular, both the communication and the coordination state are explicitly represented, from the static and the dynamic viewpoints. Since the tuple centre model is *inherently able* to perform operations (reactions) that are *invisible* to the interacting entities, there is a degree of openness that makes it possible to add observation inside the communication media *without affecting the communication level* itself – that is, *(i)* no extra operations must be added in order to achieve observability, and *(ii)* all communication events are *fully* observable, whether they change the tuple space state or not.

This feature provides a fundamental new tool, enabling us to monitor, debug and test interaction in a configurable way. So, it is possible to implement virtually any (computable) debugging policy – possibly, providing some frequent debugging rules as recurring debugging patterns. In particular, since any condition can in principle be intercepted by some suitable reaction, and handled accordingly, tuple centre-based debugging suffers no a-priori limitations. This is a much more flexible approach than many commercial packages': TupleScope by Scientific Computing's Linda implementation (described in Linda user manual), for instance, allows only a pre-defined set of conditions to be captured and monitored – those that TupleScope's designers judged to deserve such "special treatment".

We should finally remark, however, that observation problems, while prevented at the communication level, obviously re-appear at the meta-level, in both the static and the dynamic views. In fact, the set S of specification tuples can be observed statically through some ad hoc primitive – indeed, with the same limits discussed above for the tuple space, showing that observation problems have just moved one level up. From the dynamic viewpoint, tuple centres enable the set Z of triggered (both failed and successful) reactions to be observed, along with the current state of the tuple centre virtual machine: as expected, the probe problem is still there – we can add reactions to help debugging the existing reactions, but cautions have to be taken again in order to avoid interferences, since all actions occur at the same (meta) level.

[2]For details on event observability and coordination primitive semantics, we forward the interested reader to (Denti & Omicini 1999*a*)

3 Technology and Tools

The tuple centre model is the core of the TuCSoN and LuCe coordination infrastructures for multi-agent systems (Omicini & Zambonelli 1999, Denti & Omicini 1999*b*): both are based on Java as the standard "glue technology", are built upon the tuple centre notion, and focus on the interaction aspects of a MAS: so, they are orthogonal – and therefore unbounded – to any specific agent model or platform. Where the two infrastructures differ is in the abstraction of the network topology: TuCSoN promotes network-awareness, providing an articulated topology model designed to support security and mobility issues (Omicini & Zambonelli 1999, Cremonini, Omicini & Zambonelli 1999), whereas LuCe is network-transparent and specially suited for intranet MAS (Denti & Omicini 1999*b*). So, though in the following we refer mainly to TuCSoN, analogous considerations hold for LuCe, too.

3.1 Installation

The adoption of the Java 2 platform as the implementation technology makes the installation of the TuCSoN / LuCe infrastructures trivial: all is needed is to download the `tucson.jar` file to some directory of your choice. Then, a TuCSoN node can be activated (Figure 1). By default, the local node starts on the TuCSoN standard port 20504, but a different port can be specified on the command line: the TuCSoN services can then be accessed via either the provided IDE tools, or the API available in both the Java and Prolog environments.

To test whether a TuCSoN node is running on a given host, one can simply query the selected host by means of a telnet client (for instance, via a command like `telnet localhost 20504`): if present, the TuCSoN run-time (as of version 1.2.0) will reply like this:

```
TuCSoN Infrastructure (version 1.2.0)
Current date Tue Jun 18 21:29:16 CEST 2002
Running since Tue Jun 18 21:21:21 CEST 2002
```

3.2 Tools

In order to support the engineering of agent applications based on tuple centre coordination media, TuCSoN provides suitable IDE tools specifically tailored to the abstractions endorsed by the coordination infrastructure: the *Command Line Interpreter Agent* and the *Inspector*.

3.2.1 Agent Tools

The *Command Line Interpreter Agent* (also called the *CLI Agent*) tool enables users to interact with TuCSoN tuple centres anywhere on the network, in the same way as software (Java or Prolog) agents would do. A non-graphic version is also available, for devices with limited graphic support and platforms not supporting the Java Swing API (see Figure 2).

This tool lets users access tuple centres by means of Linda-like communication acts (i.e., coordination primitives) such as `out` `in` `rd`, `inp` `rdp`; moreover, they enable users to access the specification space of tuple centres via the `set_spec` and `get_spec` primitives. For instance, typing:

```
out(p(1,hello))
```

asks the infrastructure to put the tuple `p(1,hello)` in the *default* tuple centre of the *local* node. If everything goes well, the `ok` message will be displayed in reply. To read back the above tuple, an agent (either the same or another one) should issue a `rd` command whose template matches the inserted tuple, like the following:

```
rd(p(X,Y))
```

The infrastructure will then reply with a response message, providing the bindings for the `X` and `Y` variables – in this case, hopefully, something like `answer: p(1,hello)`.

To interact with other tuple centres than the default one, all is needed is to specify the desired tuple centre name in the command invocation. For instance, to insert some data in the `temperature` tuple centre, one just issues the command (Figure 2, right):

```
temperature ? out(data(jan,[15, 17.5, 14]))
```

The effect of this command is to place the tuple `data(jan,[15, 17.5, 14])` in the required tuple centre. If it does not exist yet, it is created on the fly, as if it had existed since ever to the agents' eyes (an information message is shown in the TuCSoN node console – Figure 1). Rather obviously, on-the-fly creation is performed transparently the first time a new tuple centre name is referenced, independently of the communication primitive which contains such a reference.

All the above commands apply to the LuCe infrastructure, too. Instead, the network-awareness feature is peculiar to TuCSoN: so, in the TuCSoN context, the coordination primitives can also address remote tuple centres, identified by their full name of the form `TupleCentreName @ TupleCentreIPAddress`. Here are some examples:

```
temperature @ 137.204.192.175 ? out(data(jan,[15, 17.5, 14]))
default @ bongo.unibo.it ? out(p(1))
folder(5) @ lia.org.unibo.it ? inp(guests(List))
```

Now, suppose we want to change the default behaviour of a tuple centre, re-programming it so that each inserted tuple is automatically backed up (transparently to user agents). This can be done by adding, via `set_spec`, a suitable set of `reaction` tuples that perform the required backup operation, like the following (Figure 3, top):

```
temperature ? set_spec('reaction(out(X),(out_r(backup(X)))).')
```

If the specification is accepted, you should get back the specification itself as an answer. So, all is needed to test the new behaviour is to perform an `out` operation – for instance:

```
temperature ? out(array(index(1), value(tomato)))
```

To check whether the tuple centre behaved as expected, the agent needs just to read a `backup` tuple:

```
temperature ? rd(backup(X))
```

hopefully getting `backup(array(index(1),value(tomato)))` as answer (Figure 3, bottom).

3.2.2 The Inspector Tool

The *Inspector* is a unique TuCSoN tool that enables the observation, inspection and debugging of the relevant information and activities related to the tuple centre communication and coordination states.

The key point here is that interaction between the Inspector and the observed tuple centre occurs at a different level – the infrastructure (*meta-*)level – with respect to the agent / tuple centre interaction. So, the state of communication between that tuple centre and agents can be observed from the meta-level without incurring in the probe problem.

Practically, an inspector can be spawned in two ways: either *(i)* by means of the *Inspector Spawner* application, which provides a GUI to specify the name and the address of the tuple centre to be inspected; or *(ii)* by providing such information directly at the command prompt (Figure 4). The Inspector allows users to inspect:

1. the set of tuples (button '*tuples*')
2. the set of pending queries (button '*pending*')
3. the set of behaviour specifications (button '*specification*')

4. the set of triggered reactions (button '*reactions*')

The *tuple view* (Figure 5) enables the dynamic observation of, and the action on, the set T of the logic tuples stored in the tuple centre (see Subsection 2.3). The full content of the tuple set can be replaced in one shot, by editing the list of displayed tuples: in Figure 5 (top right) this feature is exploited to add some tuples to the tuple set. Observation can be both *real-time* and *on demand*: in the first case, each change occurring in the set is immediately reflected in the tuple view, while in the second case a snapshot of the tuple set is taken only when explicitly requested via the Inspector's GUI (Figure 5, top left). It is also possible to filter the displayed tuples by specifying a tuple template (Figure 5, bottom left), as well as to log and save the (possibly filtered) tuple set shown (Figure 5, bottom right), so as to track the evolution as a sequence of tuple set snapshots. Analogously, the *pending query view* (Figure 6, top left) shows the set W of the pending `in` and `rd` queries, along with the related information such as the requesting agent, and the (tuple centre local) time when the request was received. Filtering and logging of the set content work as above.

The *specification view* (Figure 6, right) shows the set S of the current reactions: the tool includes an editor with syntax highlighting and syntax checking (button 'check'). It is also possible to load a set of pre-defined reactions, and, conversely, to save the current reaction set to a file. Once the desired reactions have been typed in or loaded, the 'accept' button sets them as the currently-valid specifications, replacing any previous set.

Finally, the *triggered reactions view* shows the set Z of the currently triggered reactions. This ability is crucial to debug coordination activities, since tracking successful and failed reactions simplifies the understanding of the dynamics of the coordination flow inside the tuple centre. For the sake of concreteness, Figure 6 (bottom left) refers to the reaction specifications shown aside and reported here below, in the case that an agent first outputs a `p(4)` tuple and then another agents emits a `q(5)` tuple:

```
reaction(out(p(X)),( in_r(p(X)), in_r(q(Y)), out_r(pq(X,Y)) )).
reaction(out(q(Y)),( in_r(p(X)), in_r(q(Y)), out_r(pq(X,Y)) )).
```

When `p(4)` is output, the first reaction is triggered, but fails due to the absence of any `q(Y)` tuple; upon insertion of `q(5)`, the second reaction is triggered, and succeeds due to the presence of `p(4)`. The result is that atomically both `p(4)` and `q(5)` are removed and the new tuple `pq(4,5)` is inserted.

In addition, the Inspector enables the management of tuple centres as *coordination virtual machines*, providing step-by-step tracing of the machine behaviour, as traditional debuggers do. This is done by selecting the *virtual machine* tab (Figure 7) and enabling the '*step execution*' option: from there, tuple centre operations can be executed singly, pressing the '*next*' button. Each time the button is pressed, the tuple centre virtual machine processes *one* communication event *and* all the reactions it triggers (in a non-deterministic order) as single macro-step of the virtual machine. At the same time, if desired, the triggered reactions can be observed in the Inspector's triggered reaction view, thus effectively supporting distributed remote debugging and monitoring.

If, for instance, an agent performs a `rdp` operation with the 'step execution' option enabled, the tuple centre will not reply to the agent immediately: instead, it will reply only when the '*next*' button is pressed. Since the semantics of the `out` primitive is asynchronous (Omicini & Denti 2001*b*), an agent emitting a tuple has the 'ok' message shown on its console immediately, even if the `out` primitive itself is processed by the virtual machine only when the '*next*' button is pressed – as it can be easily observed via the Inspector's tuple view.

3.2.3 Some small examples

The ability to specify the reactive behaviour of the coordination medium can be exploited to define custom observation and debugging policies. In fact, the Turing-equivalence of ReSpecT

provides the flexibility to express computations for tracing, monitoring, and debugging purposes, which can be particularly useful when debugging complex, multi-stage interaction protocols involving different agent roles. In these situations, a tuple centre can be programmed so as to trace the protocol evolution, making it easier to identify faults, wrong / malicious agents' interaction behaviour, etc. For instance, the following rule monitors the communication acts of agents providing a given information (e.g., a product price) and generates the related history:

```
reaction( out(price(Info)),(
  current_agent(Id), current_time(Time),
  out_r(price_history(time(Time),who(Id),content(Info))) )).
```

The rule below, instead, handles statistical information about service requests:

```
reaction( in(service(Descr)),( pre,
  in_r(service_counter(Descr,N)), N1 is N+1,
  out_r(service_counter(Descr,N1)) )).
```

A familiar example is counting the number of hits of a web page:

```
reaction( rd(web_page(Name,Content)),(
  pre,
  in_r(page_hit_counter(Name,N)),
  N1 is N+1,
  out_r(page_hit_counter(Name,N1)) )).
```

This kind of flexibility can be very useful for infrastructure maintenance, since it simplifies both the monitoring and the detection of interaction-related problems.

Coordination laws can also be exploited to prevent interactions that are invalid / undesired in the current interaction context. For instance, a tuple representing some vital information, such as the users of a system, can be protected from unwanted removal (i.e., from ***in*** operations) by a reaction like the following:

```
reaction( in(users(UserList)),( post, out_r(users(UserList)) )).
```

In the same way, insertion of critical information can be restricted to trusted sources:

```
reaction( out(price(Who,Info)),(
   no_r(source_authorised(Who)), in_r(price(Who,Info)) )).
```

Although coordination laws can be exploited for security purposes, security issues like the above should better be faced at a different (meta)level than the coordination level (Cremonini et al. (1999) provide more details on this).

4 Case Study

Here we briefly illustrate the impact and the benefits of the Inspector tool for the development and debugging of a simple version of a well-known agent interaction protocol, the Contract Net Protocol (CNP)(Smith 1979). The CNP is one of the most common protocols for agent coordination (Huhns & Stephens 1999), generally used to solve the so-called *connection problem* – i.e., how to find the appropriate agent for a given task.

According to the terminology defined by Huhns & Stephens (1999), *Manager* agents have tasks to be solved, while *Contractor* agents are able to solve tasks. The involved coordination aspects are expressed in terms of coordination laws by the ReSpecT code shown in Table 2, while the behaviour of both Managers and Contractors is described by the pseudo-code in Table 1: agent communication acts are put in evidence. It should be clear that such a code implements a very basic CNP version, whose purpose is just to show how coordination occurs – not to actually constitute a full-fledged CNP implementation. In particular, we assume here that the protocol is used in a purely cooperative context: accordingly, the protocol is designed to minimise interactions, not to prevent interferences from malevolent/competitive agents.

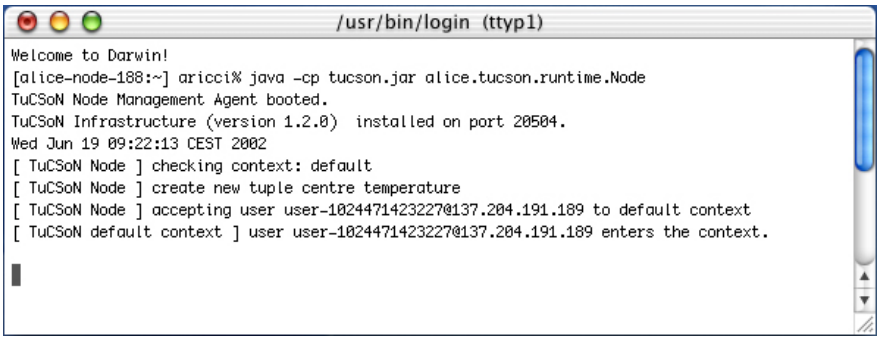
/usr/bin/login (ttyp1)
Welcome to Darwin!
[alice-node-188:~] aricci% java -cp tucson.jar alice.tucson.runtime.Node
TuCSoN Node Management Agent booted.
TuCSoN Infrastructure (version 1.2.0) installed on port 20504.
Wed Jun 19 09:22:13 CEST 2002
[TuCSoN Node] checking context: default
[TuCSoN Node] create new tuple centre temperature
[TuCSoN Node] accepting user user-10244714232270137.204.191.189 to default context
[TuCSoN default context] user user-10244714232270137.204.191.189 enters the context.

Figure 1: A TuCSoN node is started by issuing the command `java -cp tucson.jar alice.tucson.runtime.Node`. Of course, the path to the `tucson.jar` archive can be added to the `CLASSPATH` environment variable instead of using the `-cp` option. Here, the TuCSoN node also reports the creation of tuple centre `temperature`, occurred as a consequence of the operation `temperature ? out(data(jan,[15, 17.5, 14]))` discussed in the text.

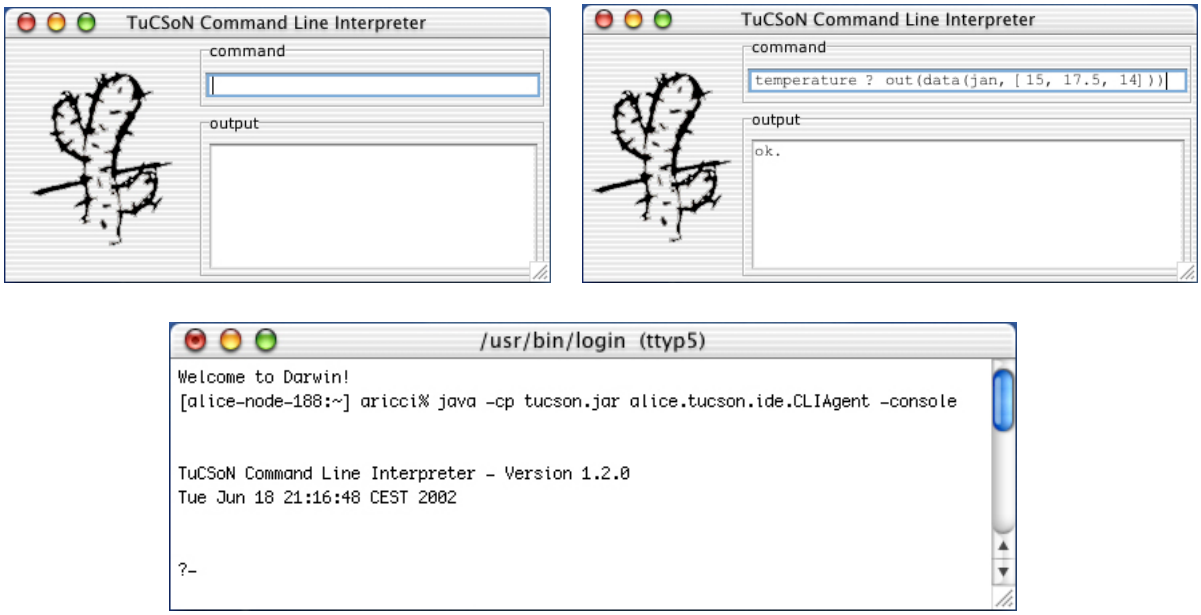

Figure 2: The TuCSoN CLI Agent: start-up *(top left)* and issuing commands *(top right)*. To start up a CLI Agent in TuCSoN 1.2.0, one just types the command `java -cp tucson.jar alice.tucson.ide.CLIAgent`: the `-console` option launches the console (non-graphic) version *(bottom)* instead of the standard graphic version.

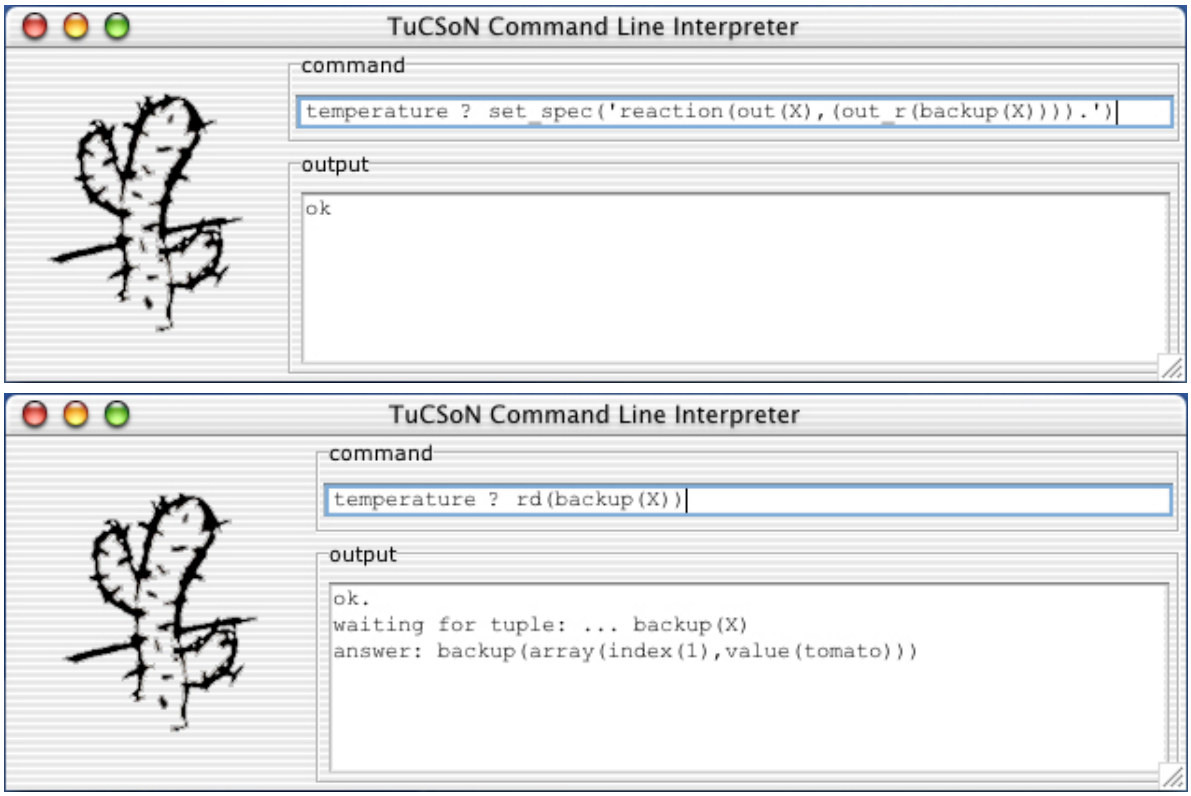

Figure 3: Re-programming the tuple centre behaviour: reaction *(top)* and effect after an `out` *(bottom)*.

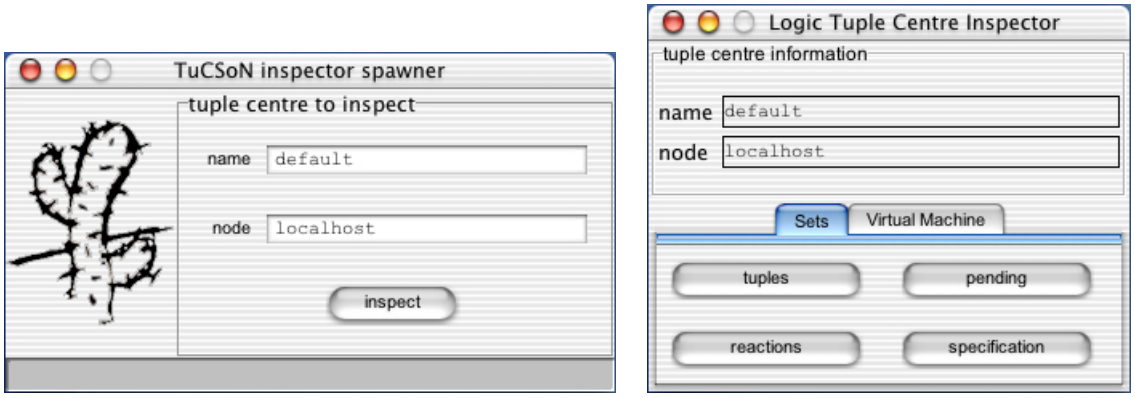

Figure 4: The Inspector Spawner *(left)* is launched issuing the command `java -cp tucson.jar alice.tucson.ide.InspectorSpawner`. Then, the Inspector tool window appears *(right)*. Alternatively, the Inspector can be activated directly by the command `javaw -cp tucson.jar alice.tucson.ide.Inspector` *`TCName TCAddress`*. Since *`TCAddress`* can be any Internet address, Inspectors can be launched on any remote node; if *`TCAddress`* is omitted, `localhost` is assumed. Here, too, the Inspector main window eventually appears.

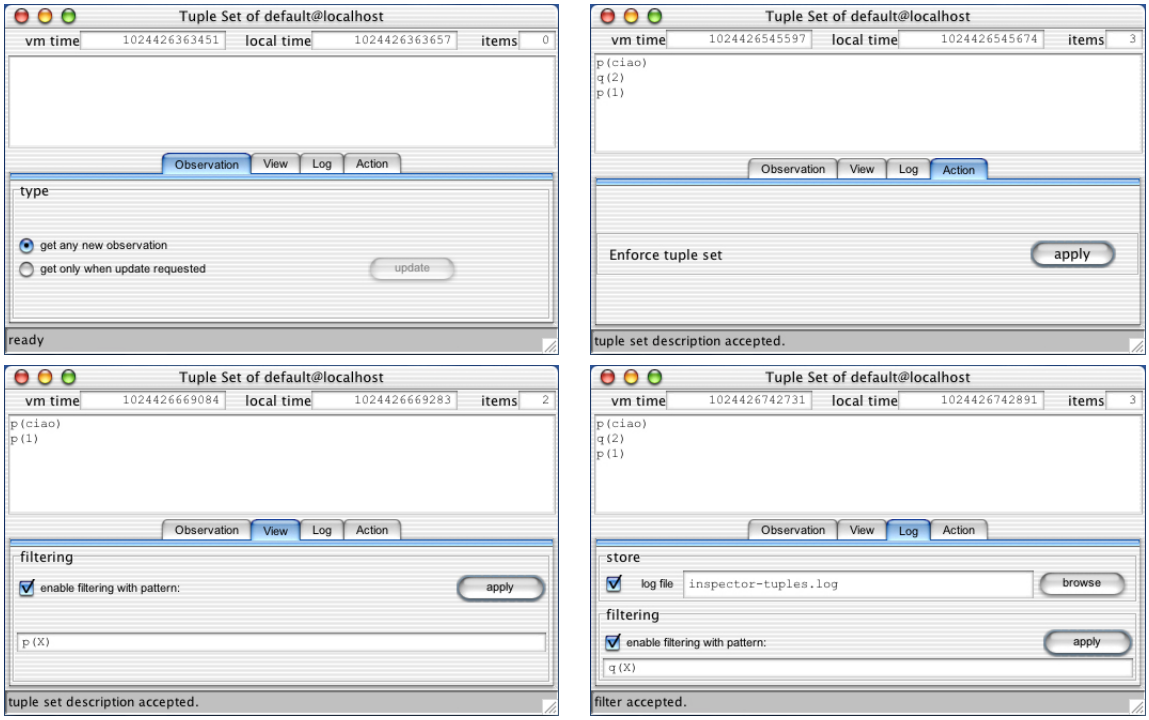

Figure 5: The Inspector *tuple view*: observation mode *(top left)*, action on the tuple set *(top right)*, filtered view of the tuple set *(bottom left)*, and log filtered view *(bottom right)*.

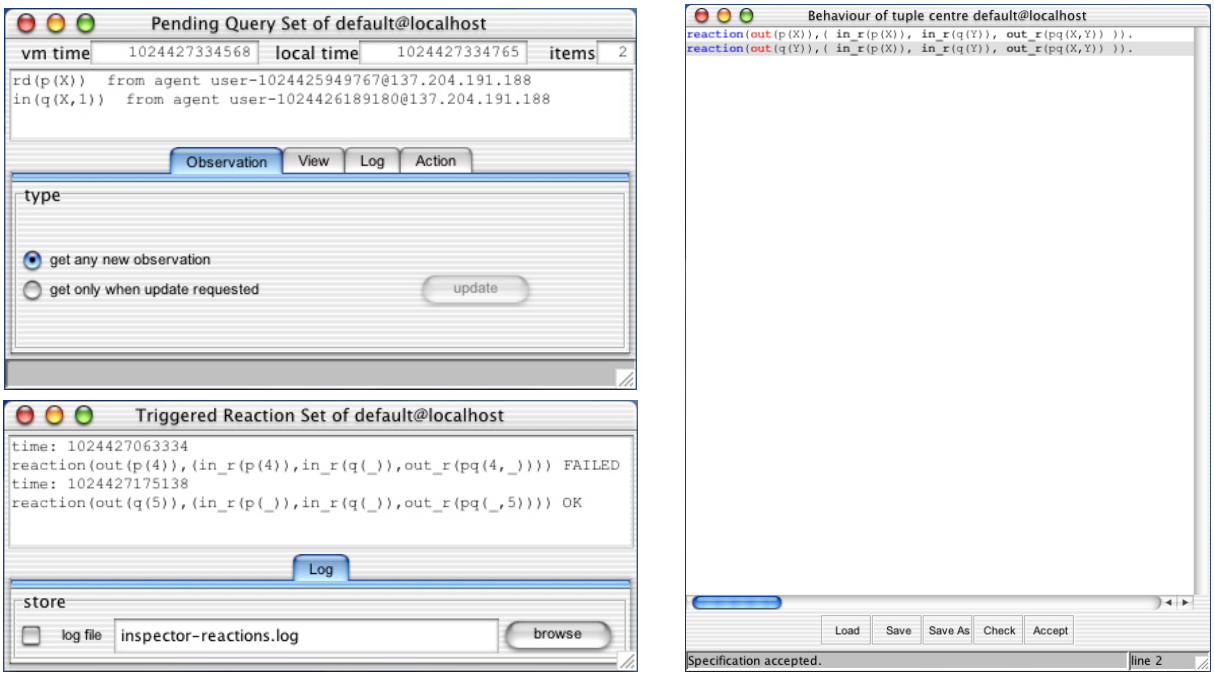

Figure 6: The Inspector *pending query view (top left)*, the *specification set (right)*, and *the triggered reactions view (bottom left)*.

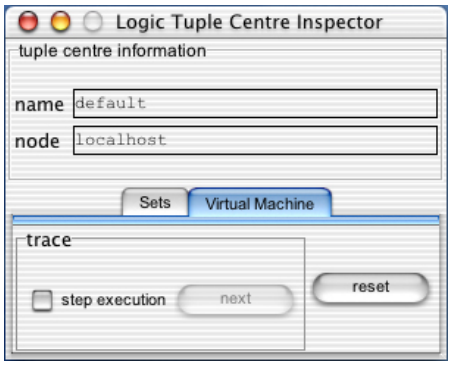

Figure 7: The Inspector tool: support for step-by-step tracing of the tuple centre virtual machine

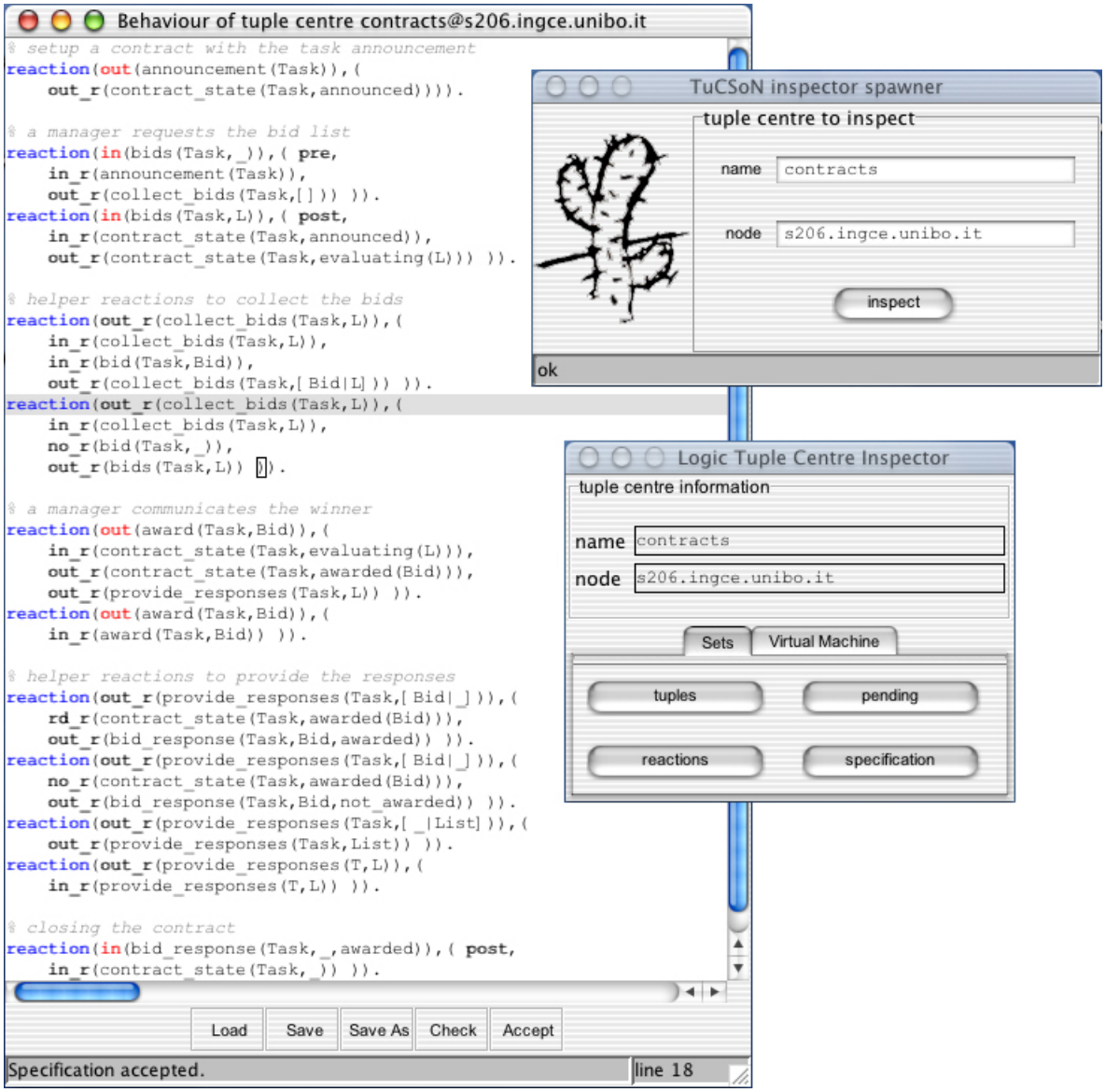

Figure 8: The Inspector tool connected to the `contracts` tuple centre and the specification view with the ReSpecT code of Table 2.

4.1 Agents Behaviour and Coordination laws in CNP

The first act of a Manager (Table 1, left) is to announce a task to be performed (line 1) and then wait for bids. When the related timeout expires, the Manager retrieves the list of bids from potential contractors (line 3), selects the best bid according to its evaluation criteria (line 4), awards the contract (line 5), and receives the result of the performed task (line 6).

On the other side, a Contractor (Table 1, right) waiting for possibly-interesting task announcements (line 1) first evaluates its capability to respond to the current request (line 2), then issues its bid (line 3), and starts waiting for an answer (line 4). Only if the bid is accepted by the Manager (answer `awarded`), the Contractor performs the task (line 5) and outputs the computed result (line 6).

The coordination rules for basic CNP are expressed by the ReSpecT code in Table 2. Note that interaction acts performed by agents (via `in`, `inp`, `out`, `rd`, `rdp`) occur separately from the coordination level of tuple centre reactions (based on `in\_r`, `out\_r`,
verb—rd_r—, `no\_r`) that specify coordination rules as algorithms operating on the tuple set.

The `contract_state` auxiliary tuple is used to track the protocol state during the coordination activity. At the beginning, when a Manager issues a task announcement, the state is set to `announced` (reaction 1). Then, when the Manager retrieves the list of proposals (meaning that the bidding stage has finished), the contract state is updated to `evaluating`, which also specifies the retrieved proposal list (reaction block 2). When the Manager eventually communicates the awarded bid, the contract state is updated accordingly, and a set of response tuples of the form `bid_response(Task,Bid,Answer)` are generated: `Answer` is `awarded` in the winning bid, and `not_awarded` in the others (reaction block 3). Finally, when the response is retrieved by the awarded bidder, the last coordination law cleans up the contract state (reaction 4).

4.2 TuCSoN Tools for CNP debugging

To show the impact of the TuCSoN support tools, let us consider the run-time inspection of the tuple centre during the enactment of the Contract Net Protocol among a set of agents. As a specific example, we consider a CNP run where the task to be assigned is represented by the tuple `purchase_book(isbn('88-06-15304-8'))`, while interacting agents are humans acting by means of the `CLIAgent` tool through the `contracts` tuple centre located at the TuCSoN node `s206.ingce.unibo.it`.

First, we use an Inspector to set the coordination specification reported in Table 2 (Figure 8): to this end, we open a specification view, write the source code (or load it from a file or paste it via copy & paste), have it syntax-checked, and upload it to the tuple centre by means of the 'accept' button.

Now, interaction starts. By means of a `CLIAgent`, a manager first makes a new announcement: its effect on the tuple centre can be observed by inspecting the triggered reaction view for the dynamics, and the tuple view for the static (final) effect Figure 9. In particular, the `out` communication action of the Manager triggers the first reaction of Table 2, whose successful execution is reported in the triggered reaction view; consequently, the tuple view reports the new tuple inserted by the Manager and the tuple generated by the reaction.

Figure 10 shows the interaction steps of a bidder. First, the bidder reads the announcement (*top*), then issues its bid (the tuple `bid(purchase_book(isbn('88-06-15304-8')),price(45))`, where `price(45)` represents the bid – *middle*), finally asks for a response (*bottom*). Figure 11 shows the state of both the tuple set and the pending query set after two more bidders have submitted a bid and are waiting for their answers: the tuple view shows the details about the submitted bids (bids with `price(45)`, `price(50)`, and `price(47)`), while the pending query view reports the requests of the three bidders waiting for an answer.

A critical point in the protocol dynamics, which is worth observing, is the closing of the bidding stage from the Manager. Figure 12 *(top)* shows the Manager's `CLIAgent` requesting a tuple

of template `bids(purchase_book(isbn('88-06-15304-8')),BidList)` and receiving the tuple `bids(purchase_book(isbn('88-06-15304-8')),[price(47),price(45),price(50)])` as result: the effect of the agent request on the tuple centre are shown in Figure 12. The triggered reaction view reports some of the reactions triggered by the request: in particular, the last reaction at the bottom of the text area is the successful reaction that inserts the tuple satisfying the manager request. That reaction is a "runtime instance" of the last reaction of block 2 in Table 2, that succeeds when no more bids can be collected.

Figure 12 shows the final effect on the tuple set of the execution of both the initial request and the subsequent reactions: no more bid tuples are present, and the tuple set includes only the tuple below, describing the new interaction state (the evaluation stage).

```
contract_state(purchase_book(isbn('88-06-15304-8')),
                    evaluating([price(47),price(45),price(50)]))
```

Another critical point in the protocol dynamics is when the manager awards the contract after collecting the bids: Figure 13 shows both the manager `CLIAgent` inserting the tuple `award(purchase_book(isbn('88-06-15304-8')),price(45))` in the tuple centre, and (parts of) the triggered reactions. In particular, the second, the fifth and the seventh reaction (belonging to the reaction block 3 of the specification shown in Table 2) insert the following response tuples for the bidders:

```
bid_response(purchase_book(isbn('88-06-15304-8')),price(45),awarded)
bid_response(purchase_book(isbn('88-06-15304-8')),price(47),not_awarded)
bid_response(purchase_book(isbn('88-06-15304-8')),price(50),not_awarded)
```

The effect of the Manager communication act on bidders is shown in Figure 14: two bidders receive a negative response, one receives the positive (awarded) response.

4.3 Debugging patterns

The programmability of the coordination medium makes it possible to specify, in principle, both general-purpose and specific *debugging patterns* as coordination laws defining the medium behaviour. To this end, the expressiveness of the specification language is the key to enable and promote a fundamental synergy between the *conceptual* tools provided by the coordination model to support the specification of the debugging patterns, and the *deployment* tools enacting and effectively supporting such patterns at run-time.

In our case, the expressiveness of the ReSpecT language makes it possible to specify any kind of debugging patterns, which can then be adapted dynamically *by need*. Moreover, the deployment tools provided, such as the Inspector, specifically exploit the synergy with the conceptual tools suggested by the ReSpecT coordination model.

With reference to out case study, let us consider the three following debugging patterns:

- *Tracing* – Protocol evolution can be explicitly documented, tracing each relevant step and easily observing the full history of the protocol interactions. For instance, the reaction block shown in Table 3 enables tracing *(i)* the announcement of a new task (reaction 1), *(ii)* the occurrence of a new bid (reaction 2), *(iii)* the announcement expiration (reaction 3), and *(iv)* the contract award (reaction 4). Each of these interaction events triggers a reaction that *reifies* the event occurrence by adding to the tuple set a tuple describing the event along with the event time-stamp and the responsible agent. In Figure 15, the top figure shows the specification view after updating the ReSpecT source with the new reactions tuples providing for tracing, while the bottom figure shows the tuple set view after a complete run of the CNP with the new coordination specification, containing history tuples about task announcement, bid submission, and contract award.

- *Statistics* – The possibility of creating (and real-time updating) statistics on the interaction dynamics can be fundamental to improve the design of coordination activities, in particular toward *time scalability* – to understand how to adapt the coordination laws to minimise interactions – and *space scalability* – to understand how to design the topology of the coordination space (number and distribution of tuple centres). For instance, the reaction block in Table 4 adds the capability to monitor the total number of announced tasks and the task rate (contracts per second) handled by a tuple centre: this information provides one first, yet coarse, estimation of the coordination burden (workload) of the tuple centre. As a further example, the reaction block in Table 5 adds the capability to trace the mean number of bids for a specific task, which can help to perform a deeper/fine-grained analysis of the coordination burden.

- *Monitoring logic events* – The capability to monitor the dynamics of interaction protocols and notify the occurrence of selected logic events is a very effective tool for debugging and testing interaction, in particular to detect logic errors in the coordination flow – in the same way as assertions support the debugging and testing of algorithms. As an example, the reaction block in Table 6 adds the capability to monitor the partial number of bids for each run of a specific task announcement: whenever such a number exceeds the threshold value stored in the `monitor_bids(Task,Threshold,NotificationCentre)` tuple (which is obviously supposed to be present in the tuple set), a warning tuple like `bids_overflow(SourceTC,Task,Threshold,N)` is inserted in the `NotificationCentre` tuple centre by an `out_tc` operation[3] thus reifying the event (reaction 4).

One first point worth remarking is that multi-agent systems that do not have interaction media as first class entities usually provide some of these debugging features as a closed set of hard-coded mechanisms, typically bounded to a specific implementation platform. Here, instead, infinite debugging strategies can be provided as programs in a Turing-equivalent interaction-oriented language (like ReSpecT) whose semantics is formally defined.

Moreover, a suitable engineering of the interaction protocol makes it possible to develop a debugging policy *incrementally*: as shown in the examples, coordination laws (here, reactions) realising debugging strategies can be progressively added to the core coordination rules without interferences. So, the encapsulation of the coordination specification provided by tuple centre is effective for engineering both the coordination activities and the debugging / testing patterns.

5 Lessons Learnt

According to our exploration, debugging is much easier and flexible when some form of mediated interaction (blackboards, tuple spaces, tuple centres, etc.) is adopted instead of point-to-point communication – whether they are low-level sockets or high-level abstractions such as KQML or FIPA ACL. Mediated interaction enables specific monitoring and debugging tools to be associated to the communication media *directly*, thus bringing the metaphors used to model the agent interaction space down to the development and deployment phases. The benefits are even more enhanced in coordination infrastructure based on tuple-based models, thanks to the uncoupling properties of generative communication and – in the case of programmable coordination media such as the tuple centres – to the ability of defining the behaviour of the communication media.

From the software engineering viewpoint, we experimented the benefits of the separation between agent computation and coordination. Interaction specifications and protocols could be tested and debugged independently of agents, either using special debugging agents, or simulating them via suitable tools – as our Inspector. In particular, the role of tools seems crucial

[3] the `out_tc(DestTC,Tuple)` operation, which inserts a tuple `Tuple` into a tuple centre `DestTC`, has been recently added to the ReSpecT primitives.

in helping engineers harnessing the intrinsic complexity of agent interaction, by providing them with the most effective views on the state and evolution over time of interaction within a MAS.

6 Conclusions and Future Work

Several proposals in literature face the issue of suitable debugging and testing tools for the deployment of agent-based systems: yet, most of them seem still focused on the notions of process and agents, trying to extend the tools found in sequential contexts toward interaction, rather than focusing on the interaction dimension *per se*. Recent works, however, outline the need of suitable infrastructure-based tools that can provide support for inspecting, monitoring and debugging interaction (Graham, McHugh, Mersic, Foster, Windly, Cleaver & Decker 2001).

In this work, we identified the key aspects of such tools and discuss their development in the context of tuple-based coordination infrastructures. The TuCSoN infrastructure, based on the ReSpecT model, and the related development and deployment tools, CLIAgent and Inspector, have been used as a reference for the discussion and for the development of the Contract Net Protocol case study. Future work includes the improvement of some services provided by Inspector tool, in particular the inspection of the dynamics inside the tuple centre virtual machine – from the observation of triggered reactions to the management of the virtual machine control flow. We expect to develop a more flexible presentation of the triggered reactions, providing different levels of detail about reaction information and filtering options. For the second issue, we expect to support the step-by-step tracing of other fine-grained interaction events, such as an imminent listening / speaking transition, as well as the interception of all the events described in tuple centre operational semantics (Omicini & Denti 2001*b*). We also plan to trace the complete execution of ReSpecT programs, showing the step-by-step execution of single reactions or ReSpecT operation (such as `out_r`): in principle, we could even provide support for undoing them. More generally, what to do on reactions at run-time – from tracing and single-step execution, to breakpoints on reactions, dynamic addition/removal of reactions transparently to the ReSpecT virtual machine (i.e., operating at the meta-level) – is a matter for further exploration.

Finally, one of the main challenge for the future is to devise new metaphors specially suited to the management and debugging of "interaction-oriented" virtual machines, going beyond the classical metaphors – traps, breakpoints, watches, etc. – of the algorithmic world.

Acknowledgements

First, we would like to thank MIUR (the Italian "Ministero dell'Istruzione, dell'Università e della Ricerca") and Nokia Research Center, Burlington, MA, USA, for partially supporting this work. Also, let us thank Paolo Petta and J.P. Müller, organisers of the Third International Symposium "From Agent Theory to Agent Implementation" (AT2AI-3), and all the attendants as well, which stimulated the further development of the concepts and ideas presented here.

References

Bates, P. (1995), 'Debugging heterogeneous distributed systems using event-based models of behaviour', *ACM Transaction on Computer Systems* **13**(1), 1–31.

Cremonini, M., Omicini, A. & Zambonelli, F. (1999), Multi-agent systems on the Internet: Extending the scope of coordination towards security and topology, *in* F. J. Garijo & M. Boman, eds, 'Multi-Agent Systems Engineering', Vol. 1647 of *LNAI*, Springer-Verlag, pp. 77–88.

```
1   out(announcement(Task))
2   wait(ExpireTime)
3   in(bids(Task,BidList))
4   Bid ← selectWinner(BidList)
5   out(award(Task,Bid))
6   in(result(Task,Result)
```

```
1   rd(announcement(Task))
2   MyBid ← evaluate(Task)
3   out(bid(Task,MyBid))
4   in(bid_response(Task,MyBid,Answer))
5   if (Answer=='awarded') {
6       Result ← perform(Task)
7       out(result(Task,Result)}
```

Table 1: Agent behaviour in the CNP: *Manager* (left) and *Contractor* (right)

```
1   reaction(out(announcement(Task)), (
      out_r(contract_state(Task,announced)))).

2   reaction(in(bids(Task,_)), (
      pre,
      in_r(announcement(Task)),
      out_r(collect_bids(Task,[])) )).
    reaction(in(bids(Task,L)), (
      post,
      in_r(contract_state(Task,announced)),
      out_r(contract_state(Task,evaluating(L))) )).

    reaction(out_r(collect_bids(Task,L)), (
      in_r(collect_bids(Task,L)),
      in_r(bid(Task,Bid)),
      out_r(collect_bids(Task,[Bid|L])) )).
    reaction(out_r(collect_bids(Task,L)), (
      in_r(collect_bids(Task,L)),
      no_r(bid(Task,_)),
      out_r(bids(Task,L)) )).

3   reaction(out(award(Task,Bid)), (
      in_r(contract_state(Task,evaluating(L))),
      out_r(contract_state(Task,awarded(Bid))),
      out_r(provide_responses(Task,L)) )).
    reaction(out(award(Task,Bid)), (
      in_r(award(Task,Bid)) )).

    reaction(out_r(provide_responses(Task,[Bid|_])), (
      rd_r(contract_state(Task,awarded(Bid))),
      out_r(bid_response(Task,Bid,awarded)) )).
    reaction(out_r(provide_responses(Task,[Bid|_])), (
      no_r(contract_state(Task,awarded(Bid))),
      out_r(bid_response(Task,Bid,not_awarded)) )).
    reaction(out_r(provide_responses(Task,[_|List])), (
      out_r(provide_responses(Task,List)) )).
    reaction(out_r(provide_responses(T,L)), (
      in_r(provide_responses(T,L)) )).

4   reaction(in(bid_response(Task,_,awarded)), (
      post,
      in_r(contract_state(Task,_)) )).
```

Table 2: ReSpecT code for the basic implementation of the Contract Net Protocol

```
1   reaction(out(announcement(Task)), (
      current_time(T),
      current_agent(Id),
      out_r(history(time(T),new_task_announcement(Task,Id))))).

2   reaction(out(bid(Task,MyBid)), (
      current_time(T),
      current_agent(Id),
      out_r(history(time(T),new_bid_arrived(Task,Id,MyBid))))).

3   reaction(in(bids(Task,L)), (
      pre,
      current_time(T),
      out_r(history(time(T),task_announcement_expired(Task))))).

4   reaction(out(award(Task,TheBid)), (
      current_time(T),
      out_r(history(time(T),contract_awarded(Task,TheBid))))).
```

Table 3: Debugging patterns, tracing: ReSpecT code to add tracing capability

```
1   reaction(out(ntasks(0)), (
      current_time(T),
      out_r(start_time(T)))).

2   reaction(out(announcement(Task)), (
      in_r(ntasks(N)),
      N1 is N + 1,
      out_r(ntasks(N1)))).

3   reaction(in(task_rate(_)), (
      pre,
      rd_r(ntasks(N)),
      rd_r(start_time(T0)),
      current_time(T1),
      DT is T1 - T0,
      Rate is N * 1000 / DT,
      out_r(task_rate(Rate)) )).
```

Table 4: Debugging patterns, statistics: ReSpecT code for computing the task announcement number and the task announcement rate

```
1   reaction(out(ntasks(Task,0)), (
      out_r(total_bids(Task,0)) )).

2   reaction(out(announcement(Task)), (
      in_r(ntasks(Task,N)),
      N1 is N + 1,
      out_r(ntasks(Task,N1)))).

3   reaction(out(bid(Task,_)), (
      in_r(total_bids(Task,N)),
      N1 is N + 1,
      out_r(total_bids(Task,N1)) )).

4   reaction(in(bids_per_task(Task,NBids)), (
      pre,
      rd_r(total_bids(Task,NBids)),
      rd_r(ntasks(Task,NTasks)),
      Value is NBids / NTasks,
      out_r(bids_per_task(Task,Value)) )).
```

Table 5: Debugging patterns, statistics: ReSpecT code for computing the mean value of bids per task

```
1   reaction(out(announcement(Task)), (
      out_r(partial_bids(Task,0)) )).

2   reaction(in_r(announcement(Task)), (
      in_r(partial_bids(Task,_)) )).

3   reaction(out(bid(Task,_)), (
      in_r(partial_bids(Task,N)),
      N1 is N + 1,
      out_r(partial_bids(Task,N1)) )).

4   reaction(out_r(partial_bids(Task,N)), (
      rd_r(monitor_bids(Task,Threshold,NotificationCentre)),
      N > Threshold,
      current_tc(SourceTC),
      out_tc(NotificationCentre,bids_overflow(SourceTC,Task,Threshold,N)))).
```

Table 6: ReSpecT code adding logic event notification capability

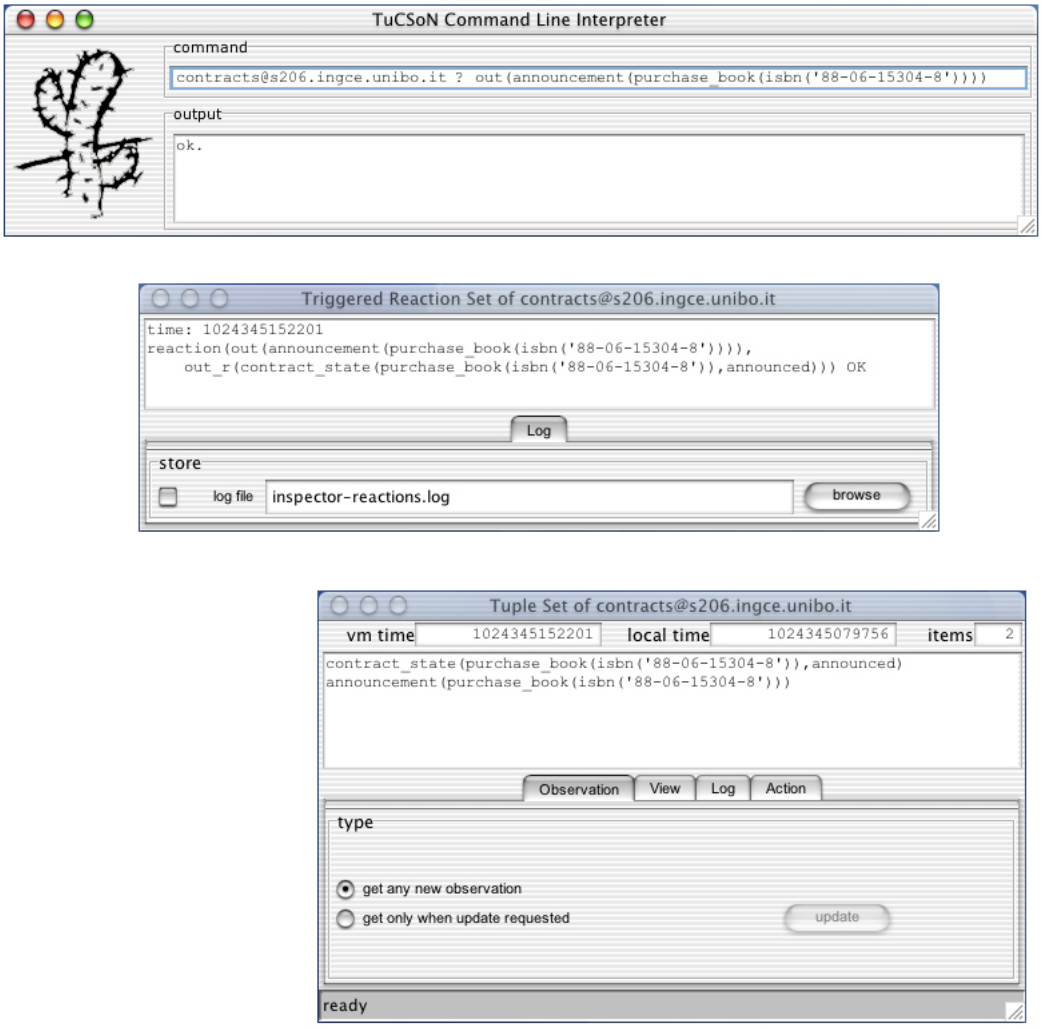

Figure 9: The `CLIAgent` of the manager announcing a new task and the related effect on the `contract` tuple centre (triggered reactions and tuple set view).

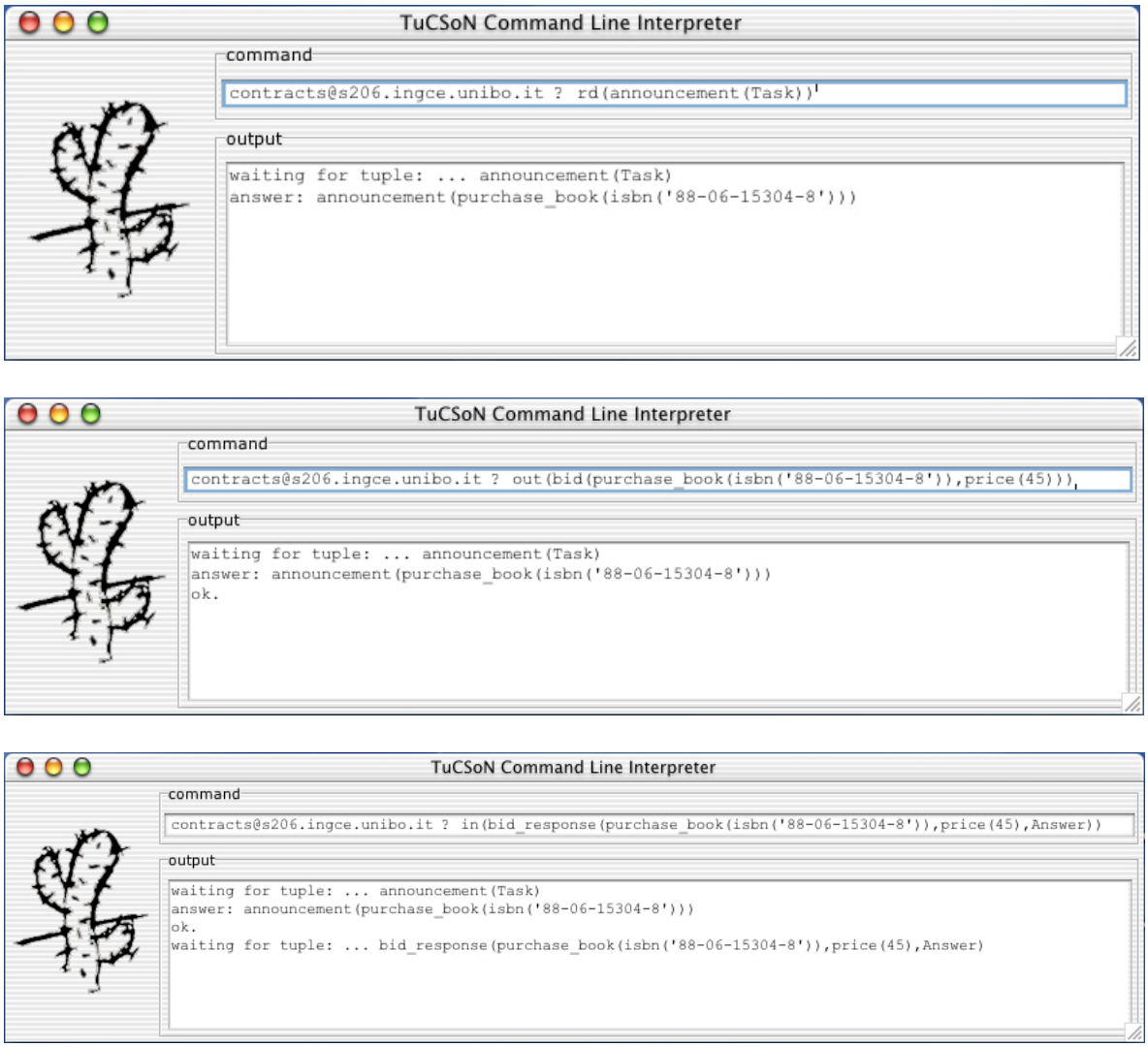

Figure 10: The `CLIAgent` of a bidder after executing some interaction steps: reading the announcement, providing a bid and waiting for the answer.

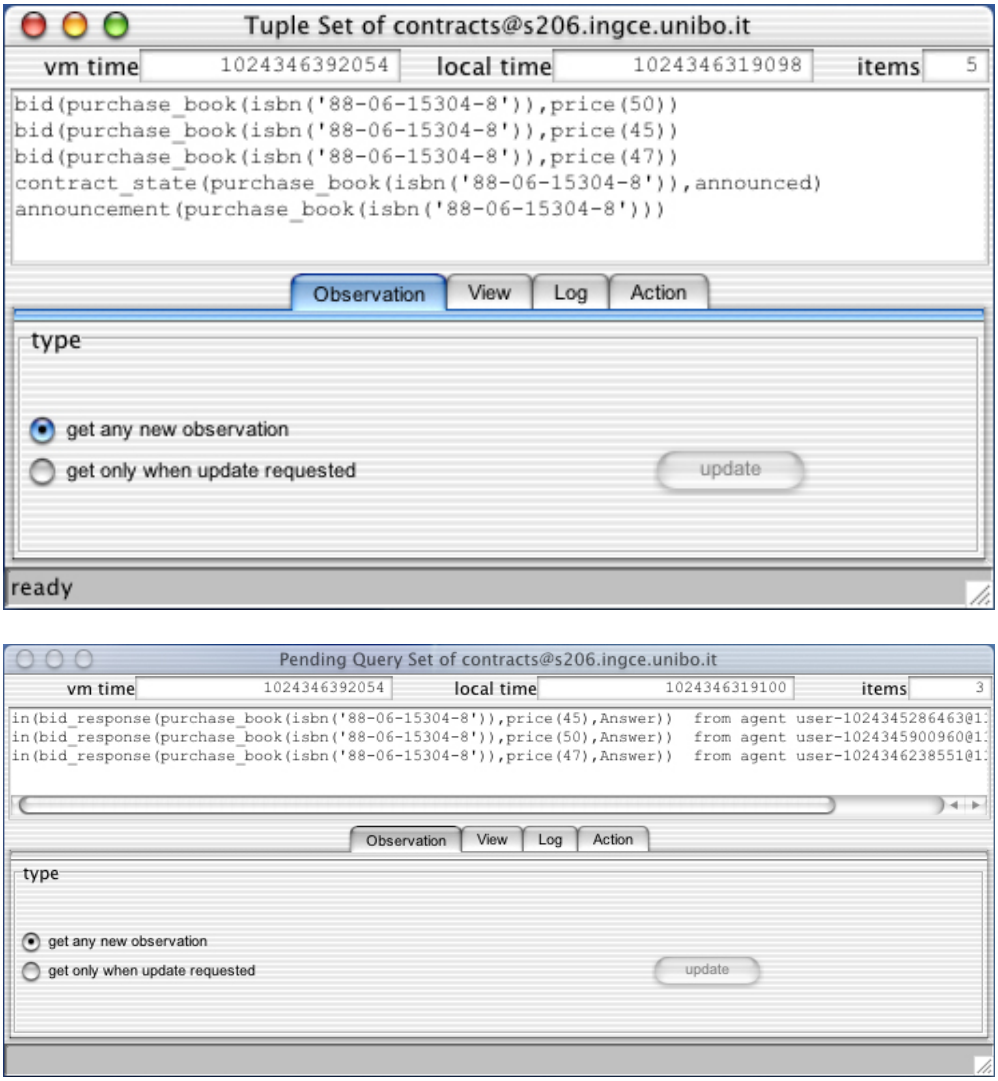

Figure 11: The tuple set view and the pending query view after the proposals of three bidders, waiting for the answer.

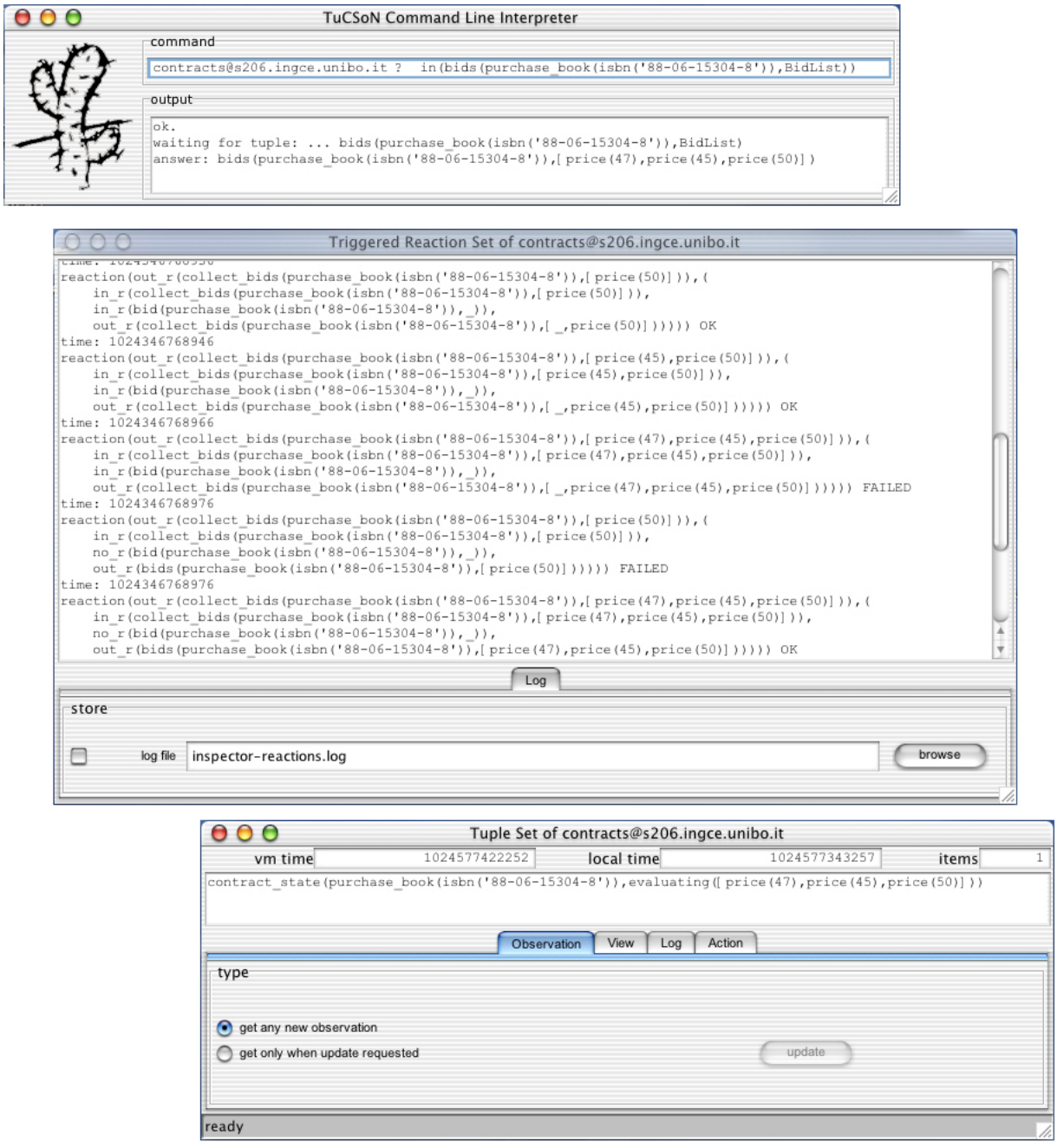

Figure 12: The `CLIAgent` of the manager requesting the bid list and the related effect on the tuple centre (triggered reactions and tuple set view).

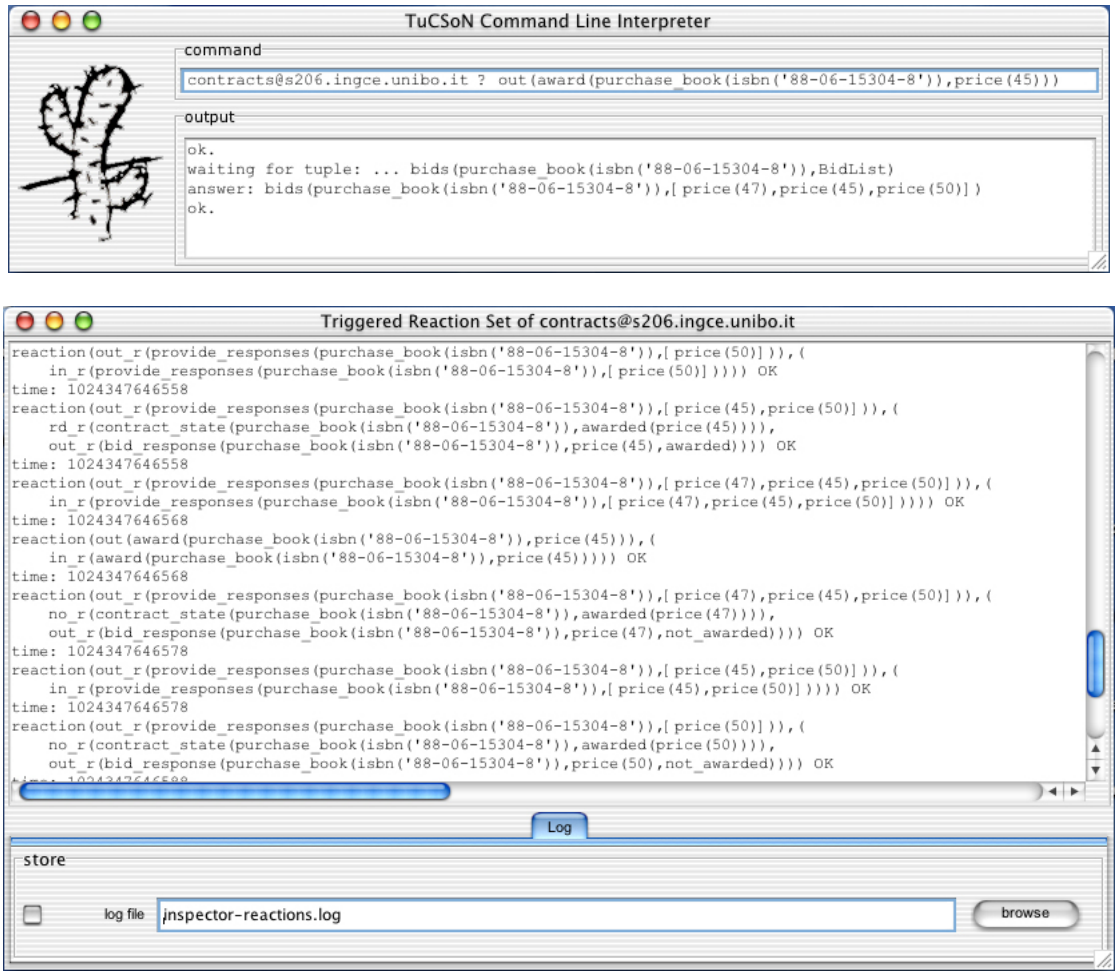

Figure 13: The `CLIAgent` of the Manager communicating the awarded bid and the part of the triggered reactions in the related view.

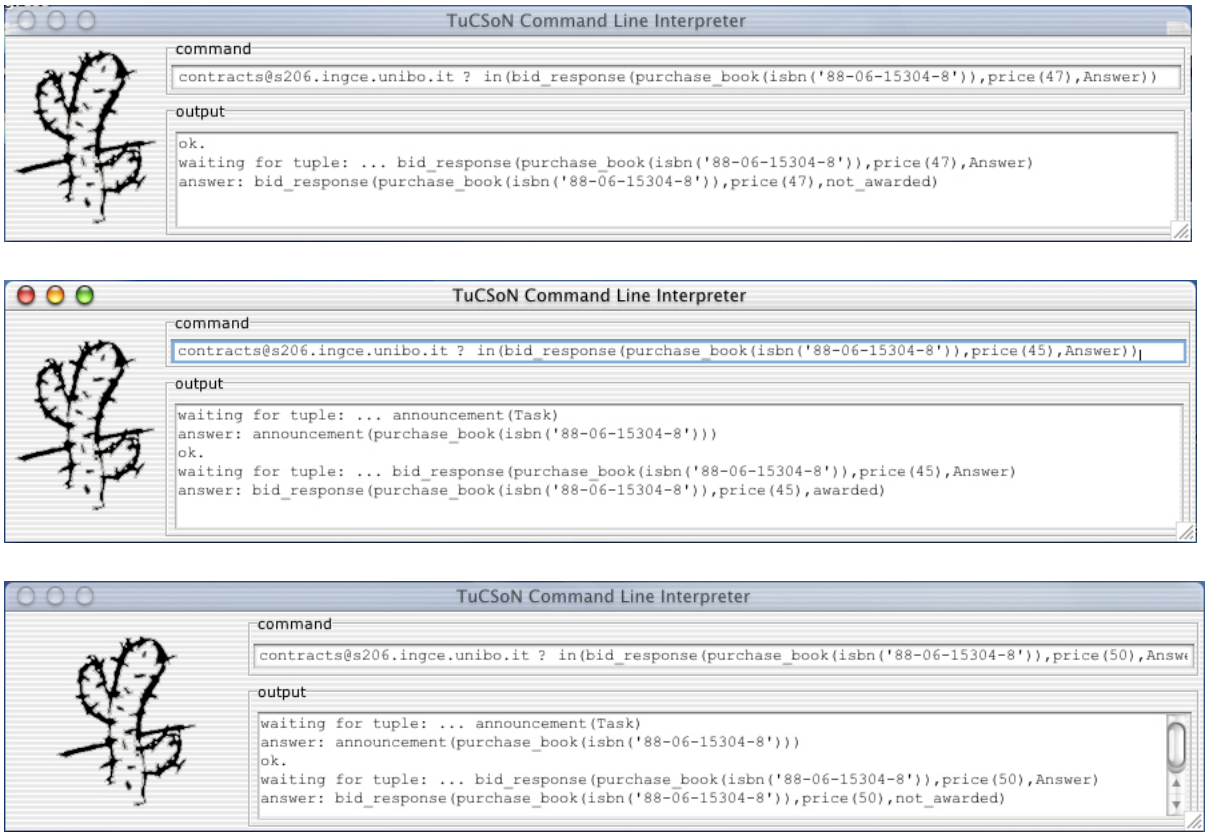

Figure 14: The `CLIAgent` of the three bidders receiving the answer to their proposal: only one is awarded (the middle one).

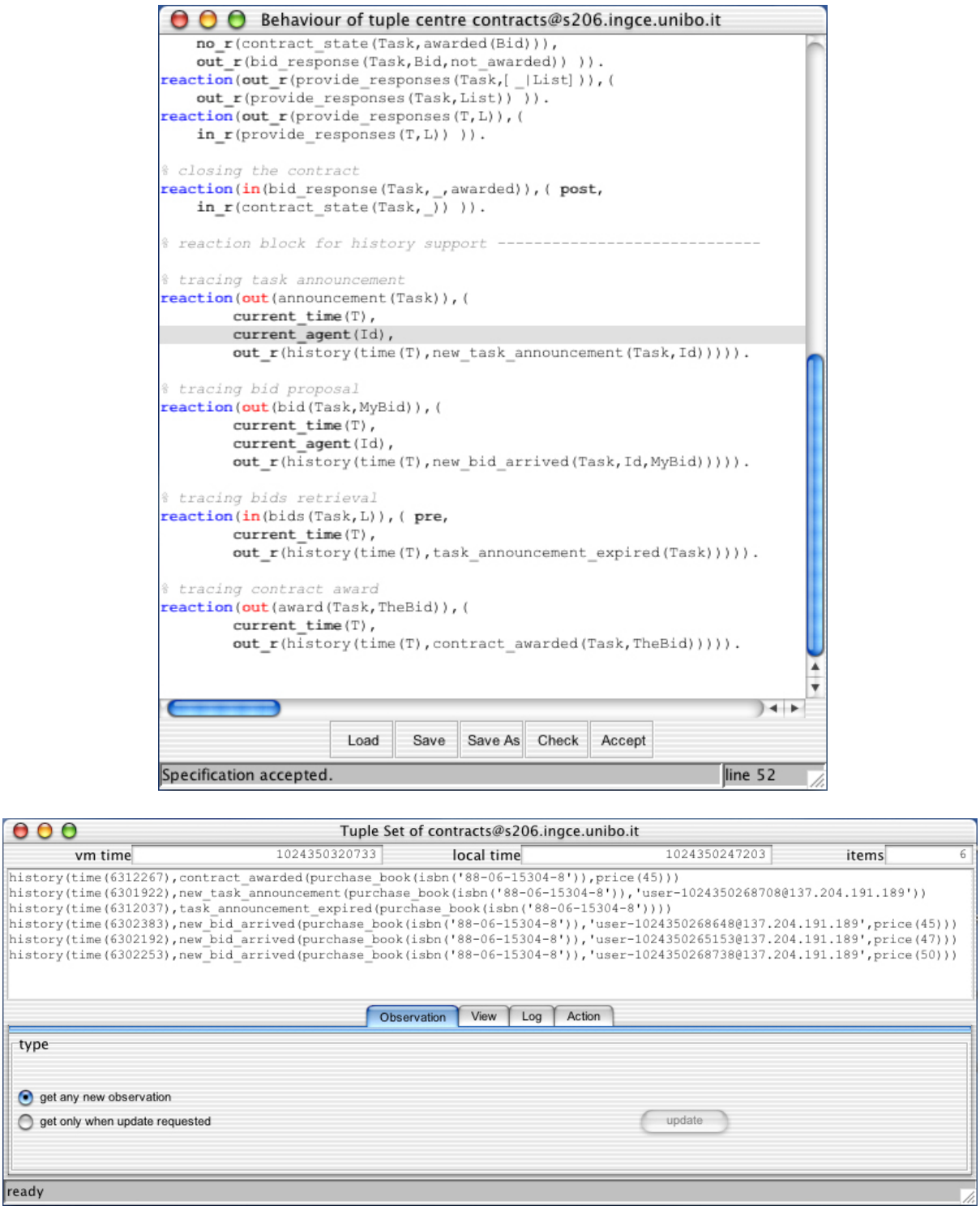

Figure 15: The specification view with the new reactions added for tracing the main interaction event *(top)* and the tuple set view after a run of the CNP with the extended specification *(bottom)*.

Denti, E. & Omicini, A. (1999*a*), 'An architecture for tuple-based coordination of multi-agent systems', *Software — Practice & Experience* **29**(12), 1103–1121.

Denti, E. & Omicini, A. (1999*b*), Engineering multi-agent systems in LuCe, *in* S. Rochefort, F. Sadri & F. Toni, eds, 'ICLP'99 International Workshop on Multi-Agent Systems in Logic Programming (MAS'99)', Las Cruces (NM).

Englemore, R. S. & Morgan, A., eds (1988), *Blackboard Systems*, Addison-Wesley.

Garcia-Molina, H., Germano, F. & Kohler, W. (1989), 'Debugging a distributed computing system', *IEEE Transaction on Software Engineering* **10**(2), 210–219.

Gelernter, D. & Carriero, N. (1992), 'Coordination languages and their significance', *Communications of the ACM* **35**(2), 97–107.

Graham, J., McHugh, D., Mersic, M., Foster, M., Windly, V., Cleaver, D. & Decker, K. (2001), Tools for developing and monitoring agents in distributed muti agent systems, *in* 'Proc. of Autonomous Agents 2000 - Workshop on Infrastructures for Scalable MASs'.

Huhns, M. & Stephens, L. (1999), Multiagent systems and societies of agents, *in* G. Weiss, ed., 'Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence', The MIT Press, pp. 79–120.

McDowell, C. & Helmbold, D. (1989), 'Debugging concurrent programs', *ACM Computing Surveys* **21**(4), 593–622.

Ndumu, D., Nwana, H., Lee, L. & Collis, J. (1999), Visualizing and debugging distributed multi-agent systems, *in* 'Proceedings of Autonomous Agents 1999', ACM.

Omicini, A. & Denti, E. (2001*a*), Formal ReSpecT, *in* A. Dovier, M. C. Meo & A. Omicini, eds, 'Declarative Programming – Selected Papers from AGP'00', Vol. 48 of *Electronic Notes in Theoretical Computer Science*, Elsevier Science B. V., pp. 179–196.

Omicini, A. & Denti, E. (2001*b*), 'From tuple spaces to tuple centres', *Science of Computer Programming* **41**(3), 277–294.

Omicini, A. & Zambonelli, F. (1999), 'Coordination for Internet application development', *Journal of Autonomous Agents and Multi-Agent Systems* **2**(3), 251–269. Special Issue: Coordination Mechanisms for Web Agents.

Rossi, D., Cabri, G. & Denti, E. (2001), Tuple-based technologies for coordination, *in* A. Omicini, F. Zambonelli, M. Klusch & R. Tolksdorf, eds, 'Coordination of Internet Agents', Springer-Verlag, chapter 4, pp. 83–109.

Rover, D., Waheed, A., Mutka, M. & Bakic, A. (1998), 'Software tools for complex distributed systems: Toward integrated tool environments', *IEEE Concurrency* **6**(2), 40–54.

Schumacher, M. (2001), *Objective Coordination in Multi-Agent System Engineering – Design and Implementation*, Vol. 2039 of *LNAI*, Springer-Verlag.

Smith, R. G. (1979), The contract net protocol: High-level communication and control in a distributed problem solver, *in* 'Proceedings of the 1st International Conference on Distributed Computing Systems', IEEE Computer Society, Washington D.C., pp. 186–192.

Van Liederkerke, M. & Avouris, N. (1995), 'Debugging multi-agent systems', *Information and Software Technology* **37**(2), 103–112.

Wegner, P. (1996), Coordination as constrained interaction, *in* P. Ciancarini & C. Hankin, eds, 'Proceedings of the 1st International Conference on Coordination Languages and Models', Vol. 1061 of *LNCS*, Springer-Verlag, pp. 28–33.

Wegner, P. (1997), 'Why interaction is more powerful than computing', *Communications of the ACM* **40**(5), 80–91.